Proceedings of the 2018 Winter Simulation Conference
M. Rabe, A. A. Juan, N. Mustafee, A. Skoogh, S. Jain, and B. Johansson, eds.

MONTE CARLO TREE SEARCH: A TUTORIAL

Michael C. Fu

Robert H. Smith School of Business, Van Munching Hall
Institute for Systems Research, A. V. Williams Building
University of Maryland
College Park, MD 20742, USA

ABSTRACT

Monte Carlo tree search (MCTS) is a general approach to solving game problems, playing a central role in Google DeepMind's AlphaZero and its predecessor AlphaGo, which famously defeated the (human) world Go champion Lee Sedol in 2016 and world #1 Go player Ke Jie in 2017. Starting from scratch without using any domain-specific knowledge (other than the game rules), AlphaZero defeated not only its ancestors in Go but also the best computer programs in chess (Stockfish) and shogi (Elmo). In this tutorial, we provide an introduction to MCTS, including a review of its history and relationship to a more general simulation-based algorithm for Markov decision processes (MDPs) published in a 2005 Operations Research article; a demonstration of the basic mechanics of the algorithms via decision trees and the game of tic-tac-toe; and its use in AlphaGo and AlphaZero.

1 INTRODUCTION

Monte Carlo tree search (MCTS) was first used by Rémi Coulom (Coulom 2006) in his Go-playing program, Crazy Stone. Prior to that, almost all of the Go-playing programs used deterministic search algorithms such as those employed for chess-playing algorithms. Google DeepMind's AlphaGo proved the wisdom and power of MCTS by employing MCTS to train its two deep neural networks, culminating in AlphaGo's resounding victories over the reigning European Go champion Fan Hui in October 2015 (5-0), the reigning world Go champion Lee Sedol in March 2016 (4-1; Seoul, South Korea), and world #1 Go player Ke Jie in May 2017 (3-0; Wuzhen, China); there is even a movie documentary about AlphaGo's amazing story (AlphaGo Movie 2018). AlphaGo started training its neural networks using past games, i.e., supervised learning, and then went on to play against itself (self-play) using reinforcement learning to train its neural networks. In contrast, both AlphaGo Zero and AlphaZero train their respective neural networks by *self play only* using MCTS for both players (reinforcement learning), viz., "The neural network in AlphaGo Zero is trained from games of self-play by a novel reinforcement learning algorithm. In each position, an MCTS search is executed, guided by the neural network. The MCTS search outputs probabilities of playing each move." In the October 19, 2017 issue of *Nature*, Silver et al. (2017a) from DeepMind announced that AlphaGo Zero had defeated AlphaGo (the version that had beaten the world champion Lee Sedol) 100 games to 0. Then, in December of 2017, AlphaZero defeated the world's leading chess playing program, Stockfish, decisively. As mentioned earlier, AlphaZero accomplished this without any (supervised) training using classical chess openings or end-game strategies, i.e., using only MCTS training.

The basic setting is that of sequential decision making under uncertainty. Generally, the objective is to maximize some expected reward at an initial state. In a game setting, the simplest version would be to maximize the probability of winning from the beginning of the game. Ideally, you would like to have a strategy that would take care of every possible situation, but for games like Go, this is impossible to achieve (see the latter section regarding Go). So, less ambitiously, you would like to choose a good first move at least. How do you decide what is a good first move? If you had an oracle, you would simply ask it to tell you the best move. However, now assume that due to bounded rationality, the oracle cannot tell you for sure what is the best move, but only give you probabilities of winning given each feasible first move. Again, if these probabilities were known precisely, you would simply take the one with the maximum probability, but in our setting these probabilities are only estimates, which can be improved by simulating more games. Thus, the objective becomes which move to simulate next to give you a better estimate of win probabilities. Since this is done sequentially, there is a tradeoff between going deeper down the tree (to the very end of the game, for instance) or trying more moves.

To make this more precise (and clearer, I hope), I will illustrate MCTS through two simple examples: decision trees and the game of tic-tac-toe. I will also demonstrate how MCTS is based on the adaptive multi-stage sampling algorithm of Chang et al. (2005).

The remainder of this tutorial will cover the following: description of MCTS illustrated using the two simple examples of decision trees and tic-tac-toe; historical roots of MCTS, including its connection to simulation-based algorithms for MDPs (Chang et al. 2007); and overview of the use of MCTS in AlphaGo and AlphaZero. Much of this material is borrowed (verbatim at times, along with the figures) from two of the author's previous expositions: A 2017 INFORMS Tutorial chapter (Fu 2017) and a WSC 2016 proceedings article (Fu 2016); see also the October 2016 *OR/MS Today* article (Chang et al. 2016).

2 MONTE CARLO TREE SEARCH: OVERVIEW AND EXAMPLES

2.1 Background

Rémi Coulom appears to be the first to use the term "Monte Carlo tree search" (MCTS), introduced in a conference paper presented in 2005/2006 (see also the Wikipedia "Monte Carlo tree search" entry), where he writes the following:

> "Our approach is similar to the algorithm of Chang, Fu and Marcus [sic] ... In order to avoid the dangers of completely pruning a move, it is possible to design schemes for the allocation of simulations that reduce the probability of exploring a bad move, without ever letting this probability go to zero. Ideas for this kind of algorithm can be found in ... *n*-armed bandit problems, ... (which) are the basis for the ***Monte-Carlo tree search*** [emphasis added] algorithm of Chang, Fu and Marcus [sic]." (Coulom 2006, p. 74)

The algorithm that is referenced by Coulom is an adaptive multi-stage simulation-based algorithm for MDPs that appeared in *Operations Research* (Chang et al. 2005), which was developed in 2002, presented at a Cornell University colloquium in the School of Operations Research and Industrial Engineering on April 30, and submitted to *Operations Research* shortly thereafter (in May). Coulom used MCTS in the computer Go-playing program that he developed, called Crazy Stone, which was the first such program to show promise in playing well on a severely reduced size board (usually 9 x 9 rather than the standard 19 x 19). Current versions of MCTS used in Go-playing algorithms are based on a version developed for games called UCT (Upper Confidence Bound 1 applied to trees) (Kocsis and Szepesvári 2006), which also references the simulation-based MDP algorithm in Chang et al. (2005) as follows:

> "Recently, Chang et al. also considered the problem of selective sampling in finite horizon undiscounted MDPs. However, since they considered domains where there is little hope that the same states will be encountered multiple times, their algorithm samples the tree in a depth-first, recursive manner: At each node they sample (recursively) a sufficient number of samples to compute a good approximation of the value of the node. The subroutine returns with an approximate evaluation of the value of the node, but the returned values are not stored (so when a node is revisited, no information is present about which actions can be expected to perform better). Similar to our proposal, they suggest to propagate the average values upwards in the tree and sampling is controlled by upper-confidence bounds. They

> prove results similar to ours, though, due to the independence of samples the analysis of their algorithm is significantly easier. They also experimented with propagating the maximum of the values of the children and a number of combinations. These combinations outperformed propagating the maximum value. When states are not likely to be encountered multiple times, our algorithm degrades to this algorithm. On the other hand, when a significant portion of states (close to the initial state) can be expected to be encountered multiple times then we can expect our algorithm to perform significantly better." (Kocsis and Szepesvári 2006, p. 292)

Thus, the main difference is that in terms of algorithmic implementation, UCT takes advantage of the game structure where board positions could be reached by multiple sequences of moves.

A high-level summary of MCTS is given in the abstract of a 2012 survey article, "A Survey of Monte Carlo Tree Search Methods":

> "Monte Carlo Tree Search (MCTS) is a rec ently proposed search method that combines the precision of tree search with the generality of random sampling. It has received considerable interest due to its spectacular success in the difficult problem of computer Go, but has also proved beneficial in a range of other domains." (Browne et al. 2012, p. 1)

The same article later provides the following overview description of MCTS:

> "The basic MCTS process is conceptually very simple... A tree is built in an incremental and asymmetric manner. For each iteration of the algorithm, a tree policy is used to find the most urgent node of the current tree. The tree policy attempts to balance considerations of exploration (look in areas that have not been well sampled yet) and exploitation (look in areas which appear to be promising). A simulation is then run from the selected node and the search tree updated according to the result. This involves the addition of a child node corresponding to the action taken from the selected node, and an update of the statistics of its ancestors. Moves are made during this simulation according to some default policy, which in the simplest case is to make uniform random moves. A great benefit of MCTS is that the values of intermediate states do not have to be evaluated, as for depth-limited minimax search, which greatly reduces the amount of domain knowledge required. Only the value of the terminal state at the end of each simulation is required." (Browne et al. 2012, pp. 1-2)

MCTS consists of four main "operators", which here will be interpreted in both the decision tree and game tree contexts:

- "Selection" corresponds to choosing a move at a node or an action in a decision tree, and the choice is based on the upper confidence bound (UCB) (Auer et al. 2002) for each possible move or action, which is a function of the current estimated value (e.g., probability of victory) plus a "fudge" factor.
- "Expansion" corresponds to an outcome node in a decision tree, which is an opponent's move in a game, and it is modeled by a probability distribution that is a function of the state reached after the chosen move or action (corresponding to the transition probability in an MDP model).
- "Simulation/Evaluation" corresponds to returning the estimated value at a given node, which could correspond to the actual end of the horizon or game, or simply to a point where the current estimation may be considered sufficiently accurate so as not to require further simulation.
- "Backup" corresponds to the backwards dynamic programming algorithm employed in decision trees and MDPs.

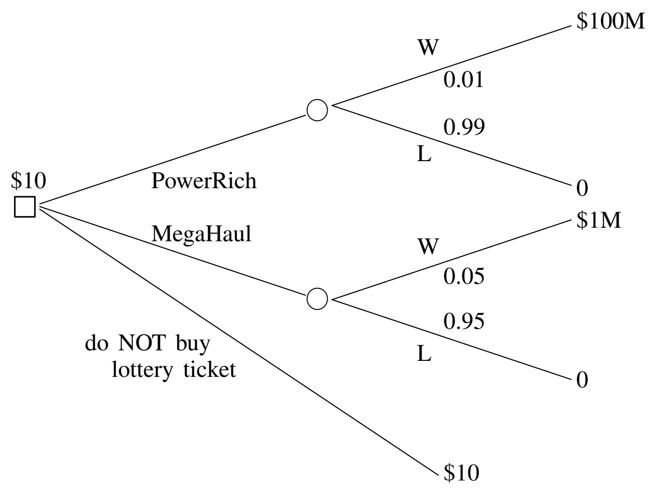

Figure 1: Decision tree for lottery choices.

2.2 Example: A Simple Decision Tree

We begin with a simple decision problem. Assume you have \$10 in your pocket, and you are faced with the following three choices: 1) buy a PowerRich lottery ticket (win \$100M w.p. 0.01; nothing otherwise); 2) buy a MegaHaul lottery ticket (win \$1M w.p. 0.05; nothing otherwise); 3) do not buy a lottery ticket.

In Figure 1, the problem is represented in the form of a decision tree, following the usual convention where squares represent decision nodes and circles represent outcome nodes. In this one-period decision tree, the initial "state" is shown at the only decision node, and the decisions are shown on the arcs going from decision node to outcome node. The outcomes and their corresponding probabilities are given on the arcs from the outcome nodes to termination (or to another decision node in a multi-period decision tree). Again, in this one-period example, the payoff is the value of the final "state" reached.

If the goal is to have the most money, which is the optimal decision? Aside from the obvious expected value maximization, other possible objectives include:

- Maximize the probability of having enough money for a meal.
- Maximize the probability of not being broke.
- Maximize the probability of becoming a millionaire.
- Maximize a quantile.

It is also well known that human behavior does not follow the principle of maximizing expected value. Recalling that a probability is also an expectation (of the indicator function of the event of interest), the first three bullets fall into the same category, as far as this principle goes. Utility functions are often used to try and capture individual risk preferences, e.g., by converting money into some other units in a nonlinear fashion. Other approaches to modeling risk include prospect theory (Kahneman and Tversky 1979), which indicates that humans often distort their probabilities, differentiate between gains and losses, and anchor their decisions. Recent work (Prashanth et al. 2016) has applied cumulative prospect theory to MDPs.

We return to the simple decision tree example. Whatever the objective and whether or not we incorporate utility functions and distorted probabilities into the problem, it still remains easy to solve, because there are only two real choices to evaluate (the third is trivial), and these have outcomes that are assumed known with known probabilities. However, we now ask the question(s): What if ...

- the probabilities are unknown?
- the outcomes are unknown?
- the terminal nodes keep going? (additional sequence(s) of action – outcome, etc.)

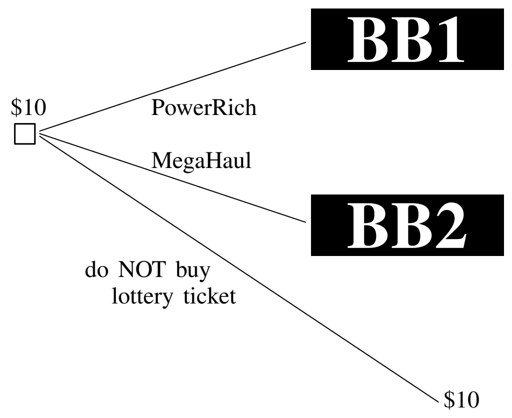

Figure 2: Decision tree for lottery choices with black boxes BB1 and BB2.

When all of these hold, we arrive at the setting where simulation-based algorithms for MDPs come into play; further, when "nature" is the opposing player, we get the game setting, for which MCTS is applied. The probabilities and outcomes are assumed unknown but can be sampled from a black box simulator, as shown in Figure 2. If there were just a single black box, we would just simulate the one outcome node until the simulation budget is exhausted, because the other choice (do nothing) is known exactly. Conversely, we could pose the problem in terms of how many simulations it would take to guarantee some objective, e.g., with 99% confidence that we could determine whether or not the lottery has a higher expected value than doing nothing ($10); again there is no allocation as to where to simulate but just when to stop simulating. However, with two random options, there has to be a decision made as to which black box to simulate, or for a fixed budget how to allocate the simulation budget between the two simulators. This falls into the domain of ranking and selection, but where the samples arise in a slightly different way, because there are generally dynamics involved, so one simulation may only be a partial simulation of the chosen black box.

2.3 Example: Tic-Tac-Toe

Tic-tac-toe is a well-known two-player game on a 3 x 3 grid, in which the objective is to get 3 marks in a row, and players alternate between the "X" player who goes first and the "O" player. If each player follows an optimal policy, then the game should always end in a tie (cat's game). In theory there are something like 255K possible configurations that can be reached, of which only an order of magnitude less of these are unique after accounting for symmetries (rotational, etc.), and only 765 of these are essentially different in terms of actual moves for both sides. The optimal strategy for either side can be easily described in words in a short paragraph, and a computer program to play optimally requires well under a hundred lines of code in most programming languages.

Assuming the primary objective (for both players) is to win and the secondary objective is to avoid losing, the following "greedy" policy is optimal (for both players):

If a win move is available, then take it; else if a block move is available, then take it.

(A win move is defined as a move that gives three in a row for the player who makes the move; a block move is defined here as a move that is placed in a row where the opposing player already has two moves, thus preventing three in a row.) This leaves still the numerous situations where neither a win move nor a block move is available. In traditional game tree search, these would be enumerated (although as we noted previously, it is easier to implement using further if–then logic). We instead illustrate the Monte Carlo tree search approach, which samples the opposing moves rather than enumerating them.

If going first (as X), there are three unique first moves to consider – corner, side, and middle. If going second (as O), the possible moves depend of course on what X has selected. If the middle was chosen, then there are two unique moves available (corner or non-corner side); if a side (corner or non-corner)

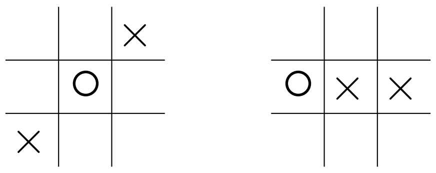

Figure 3: Two tic-tac-toe board configurations to be considered.

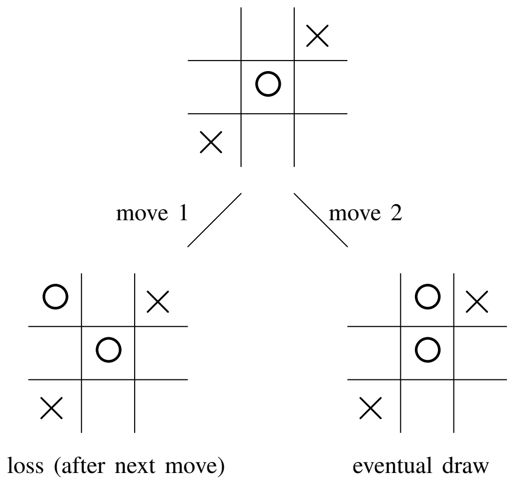

Figure 4: Game tree for 1st tic-tac-toe board configuration (assuming "greedy optimal" play), where note that if opponent's moves were completely randomized, "O" would have an excellent chance of winning!

was chosen, then there are five uniques moves (but a different set of five for the two choices) available. However, even though the game has a relatively small number of outcomes compared to most other games (even checkers), enumerating them all is still quite a mess for illustration purposes, so we simplify further by viewing two different game situations that already have preliminary moves.

Assume henceforth that we are the "O" player. We begin by illustrating with two games that have already seen three moves, 2 by our opponent and 1 by us, so it is our turn to make a move. The two different board configurations are shown in Figure 3. For the first game, by symmetry there are just two unique moves for us to consider: corner or non-corner side. In this particular situation, following the optimal policy above leads to an easy decision: corner move leads to a loss, and non-corner move leads to a draw; there is a unique "sample path" in both cases and, thus, no need to simulate. The trivial game tree and decision tree are given in Figures 4 and 5, respectively, with the game tree that allows non-optimal moves shown in Figure 6.

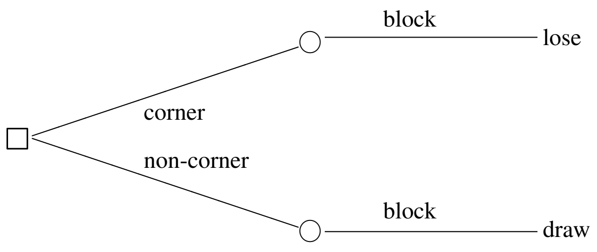

Figure 5: Decision tree for 1st tic-tac-toe board configuration of Figure 3.

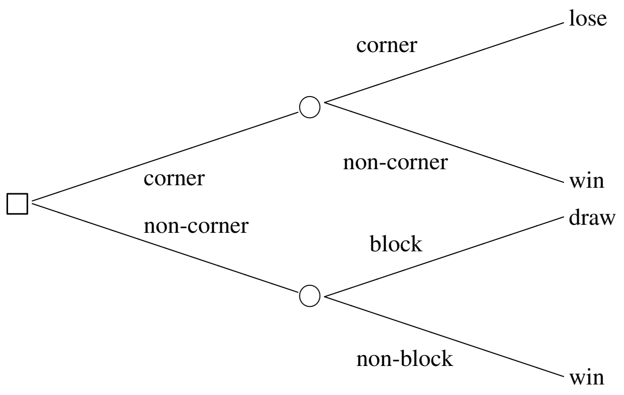

Figure 6: Decision tree for 1st tic-tac-toe board configuration with non-optimal greedy moves.

Now consider the other more interesting game configuration, the righthand side of Figure 3, where there are three unique moves available to us; without loss of generality, they can be the set of moves on the top row. We need to evaluate the "value" of each of the three actions to determine which one to select. It turns out that going in the upper left corner leads to a draw, whereas the upper middle and upper right corner lead to 5 possible unique moves for the opponent. The way the Monte Carlo tree branching would work is depicted in Figure 7, where two computational decisions would need to be made:

- How many times should each possible move (action) be simulated?
- How far down should the simulation go (to the very end or stop short at some point)?

Note that these questions have been posed in a general framework, with tic-tac-toe merely illustrating how they would arise in a simple example.

3 ADAPTIVE MULTISTAGE SAMPLING ALGORITHM

In this section, we relate MCTS to the adaptive multistage sampling (AMS) algorithm in Chang et al. (2005). Consider a finite horizon MDP with finite state space S, finite action space A, non-negative bounded reward function R such that $R : S \times A \to \mathscr{R}^+$, and transition function P that maps a state and action pair to a probability distribution over state space S. We denote the feasible action set in state $s \in S$ by $A(s) \subset A$ and the probability of transitioning to state $s' \in S$ when taking action a in state $s \in S$ by $P(s,a)(s')$. In terms of a game tree, the initial state of the MDP or root node in a decision tree corresponds to some point in a game where it is our turn to make a move. A simulation replication or sample path from this point is then a sequence of alternating moves between us and our opponent, ultimately reaching a point where the final result is "obvious" (win or lose) or "good enough" to compare with another potential initial move, specifically if the value function is precise enough.

Some questions/answers in a nutshell:

- How does AMS work?
  (1) UCB: selecting our (decision maker's) actions to simulate throughout a sample path.
  (2) simulation/sampling: generating next state transitions (nature's "actions").
- How does MCTS work?
  (1) how to select our moves to follow in a game tree.
  (2) how to simulate opponent's moves.
- How does MCTS fit into AlphaGo?
  (1) UCB: selecting our next moves in a simulated game tree path.
  (2) simulation/sampling: generating opponent's next moves.

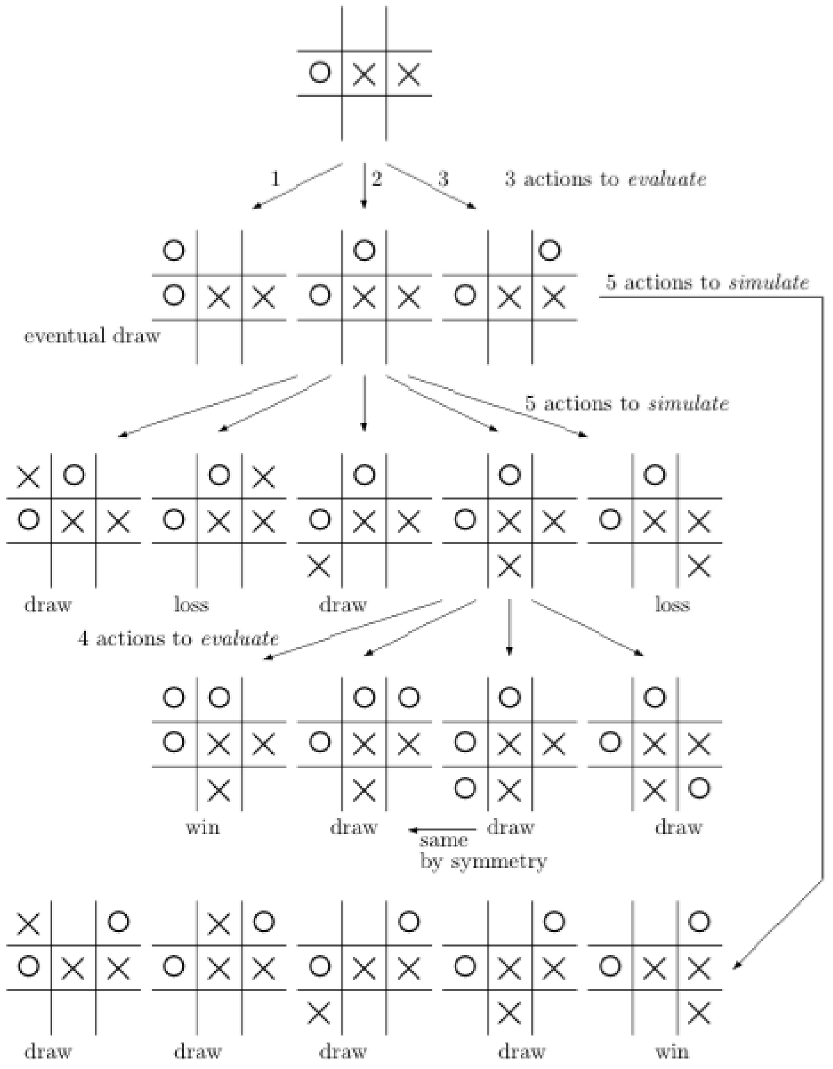

Figure 7: Monte Carlo game tree for 2nd tic-tac-toe board configuration of Figure 3.

The AMS algorithm of Chang et al. (2005) chooses to sample in state s an optimizing action in $A(s)$ according to the following:

$$\max_{a \in A(s)} \left(\hat{Q}(s,a) + \sqrt{\frac{2 \ln \bar{n}}{N_a^s}} \right), \tag{1}$$

where N_a^s is the number of times action a has been sampled thus far (from state s), $\bar{n}$ is the total number of samples thus far, and

$$\hat{Q}(s,a) = R(s,a) + \frac{1}{N_a^s} \sum_{s' \in S_a^s} \hat{V}(s'),$$

where S_a^s is the set of sampled next states thus far ($|S_a^s| = N_{a,i}^s$) with respect to the distribution $P(s,a)$, and $\hat{V}$ is the value function estimate (as a function of the state). The argument in (1) is the upper confidence bound (UCB).

We now return to the tic-tac-toe example to illustrate these concepts. In MDP notation, we will model the state as a 9-tuple corresponding to the 9 locations, starting from upper left to bottom right, where 0 will correspond to blank, 1 = X, and 2 = O; thus, (0,0,1,0,2,0,1,0,0) and (0,0,0,2,1,1,0,0,0) correspond to the states of the two board configurations of Figure 3. Actions will simply be represented by the 9 locations, with the feasible action space the obvious remainder set of the components of the space that are still 0, i.e., $\{1,2,4,6,8,9\}$ and $\{1,2,3,7,8,9\}$ for the two board configurations of Figure 3. This is not necessarily the best state representation (e.g., if we try to detect certain structure or symmetries). If the learning algorithm is working how we would like it to work ideally, it should converge to the degenerate distribution that chooses with probability one the optimal action, which can be easily determined in this simple game.

4 GO AND MCTS

Go is the most popular two-player board game in East Asia, tracing its origins to China more than 2,500 years ago. According to Wikipedia, Go is also thought to be the oldest board game still played today. Since the size of the board is 19×19, as compared to 8×8 for a chess board, the number of board configurations for Go far exceeds that for chess, with estimates at around 10^{170} possibilities, putting it a googol (no pun intended) times beyond that of chess and exceeding the number of atoms in the universe (https://googleblog.blogspot.nl/2016/01/alphago-machine-learning-game-go.html). Intuitively, the objective is to have "captured" the most territory by the end of the game, which occurs when both players are unable to move or choose not to move, at which point the winner is declared as the player with the highest score, calculated according to certain rules. Unlike chess, the player with the black (dark) pieces moves first in Go, but same as chess, there is supposed to be a slight first-mover advantage (which is actually compensated by a fixed number of points decided prior to the start of the game).

Perhaps the closest game in the Western world to Go is Othello, which like chess is played on an 8×8 board, and like Go has the player with the black pieces moving first. Similar to Go, there is also a "flanking" objective, but with far simpler rules. The number of legal positions is estimated at less than 10^{28}, nearly a googol and a half times less than the estimated number of possible Go board positions. As a result of the far smaller number of possibilities, traditional exhaustive game tree search (which could include heuristic procedures such as genetic algorithms and other evolutionary approaches leading to a pruning of the tree) can in principle handle the game of Othello, so that brute-force programs with enough computing power will easily beat any human. More intelligent programs can get by with far less computing (so that they can be implemented on a laptop or smartphone), but the point is that complete solvability is within the realm of computational tractability given today's available computing power, whereas such an approach is doomed to failure for the game of Go, merely due to the 19×19 size of the board.

Similarly, IBM Deep Blue's victory (by 3 1/2 to 2 1/2 points) over the chess world champion Garry Kasparov in 1997 was more of an example of sheer computational power than true artificial intelligence, as reflected by the program being referred to as "the primitive brute force-based Deep Blue" in the current Wikipedia account of the match. Again, traditional game tree search was employed, which becomes impractical for Go, as alluded to earlier. The realization that traversing the entire game tree was computationally infeasible for Go meant that new approaches were required, leading to a fundamental paradigm shift, the main components being Monte Carlo sampling (or simulation of sample paths) and value function approximation, which are the basis of simulation-based approaches to solving Markov decision processes (Chang et al. 2007), which are also addressed by neuro-dynamic programming (Bertsekas and Tsitsiklis 1996); approximate (or adaptive) dynamic programming (Powell 2010); and reinforcement learning (Sutton and Barto 1998). However, the setting for these approaches is that of a single decision maker tackling problems involving a sequence of decision epochs with uncertain payoffs and transitions. The game setting adapted these frameworks by modeling the uncertain transitions – which could be viewed as the actions of "nature" – as the action of the opposing player. As a consequence, to put the game setting into the MDP setting required modeling the state transition probabilities as a distribution over the actions

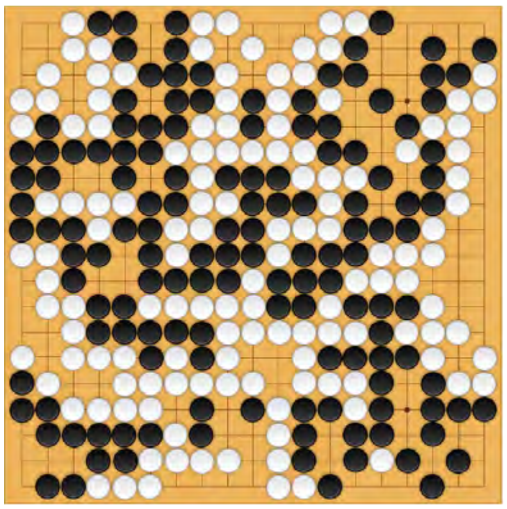

Figure 8: Final positions of Games 1 (left) and 3 (right) between AlphaGo and #1-rated human Ke Jie; AlphaGo is white in Game 1 and black in Game 3; reproduced from DeepMind Web site.

of the opponent. Thus, as we shall describe later, AlphaGo employs two deep neural networks: one for value function approximation and the other for policy approximation, used to sample opponent moves.

In March 2016 in Seoul, Korea, Google DeepMind's computer program AlphaGo defeated the reigning human world champion Go player Lee Sedol 4 games to 1, representing yet another advance in artificial intelligence (AI) arguably more impressive than previous victories by computer programs in chess (IBM's Deep Blue) and Jeopardy (IBM's Watson), due to the sheer size of the problem. A little over a year later, in May 2017 in Wuzhen, China, at "The Future of Go Summit," a gathering of the world's leading Go players, AlphaGo cemented its reputation as the planet's best by defeating Chinese Go Grandmaster and world number one (only 19 years old) Ke Jie (three games to none). To get a flavor of the complexity of the game, the final board positions of Game 1 and 3 are shown in Figure 8. In Game 1, the only one of the three played to completion, AlphaGo won by just 0.5 points playing the white pieces, with a compensation (called komi) of 7.5 points. In Game 3, Ke Jie playing white resigned after 209 moves.

An interesting postscript to the match was Ke Jie's remarks two months later (July 2017), as quoted on the DeepMind Web site blog (DeepMind 2018): "After my match against AlphaGo, I fundamentally reconsidered the game, and now I can see that this reflection has helped me greatly. I hope all Go players can contemplate AlphaGo's understanding of the game and style of thinking, all of which is deeply meaningful. Although I lost, I discovered that the possibilities of Go are immense and that the game has continued to progress."

Before describing AlphaGo's two deep neural networks that are trained using MCTS, a little background is provided about DeepMind, the London-based artificial intelligence (AI) company founded in 2010 by Demis Hassabis, Shane Legg and Mustafa Suleyman. DeepMind built its reputation on the use of deep neural networks for AI applications, most notably video games, and was acquired by Google in 2014 for $500M. David Silver, the DeepMind Lead Researcher for AlphaGo and lead author on all three articles (Silver et al. 2016; 2017a; 2017b) is quoted on the AlphaGo movie Web page (AlphaGo Movie 2017): "The Game of Go is the holy grail of artificial intelligence. Everything we've ever tried in AI, it just falls over when you try the game of Go." However, the company emphasizes that it focuses on building learning algorithms that can be generalized, just as AlphaGo Zero and AlphaZero developed from AlphaGo. In addition to deep neural networks and reinforcement learning, it develops other neuroscience-inspired models.

AlphaGo's two neural networks are depicted in Figure 9:

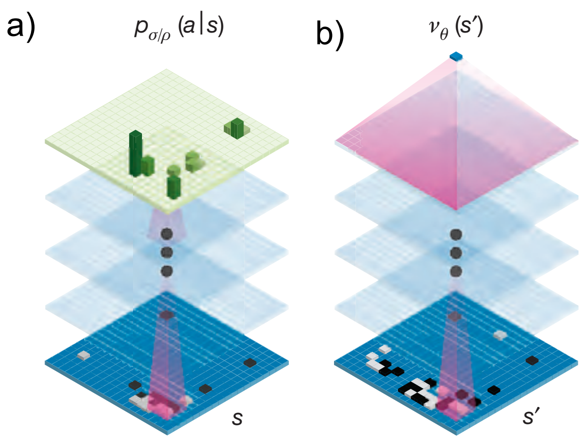

Figure 9: AlphaGo's two deep neural networks, (a) policy network; (b) value network (Adapted by permission from Macmillan Publishers Ltd.: *Nature*, Figure 1b, Silver et al. 2016, copyright 2016).

- value network: estimate the "value" of a given board configuration (state), i.e., the probability of winning from that position;
- policy network: estimate the probability distribution of moves (actions) from a given position (state).

The subscripts σ and ρ on the policy network correspond to two different networks used, using supervised learning and reinforcement learning, respectively. The subscript θ on the value network represents the parameters of the neural net. AlphaGo's two neural networks employ 12 layers with millions of connections. AlphaGo Zero has a single neural network for both the policy network and value network.

In terms of MDPs and Monte Carlo tree search, let the current board configuration (state) be denoted by s^*. Then we wish to find the best move (optimal action) a^*, which leads to board configuration (state) s, followed by (sampled/simulated) opponent move (action) a, which leads to board configuration (new "post-decision" state) s', i.e., a sequence of a pair of moves can be modeled as

$$s^* \xrightarrow{a^*} s \xrightarrow{a} s',$$

using the notation consistent with Figure 9.

As an example, consider again the tic-tac-toe example, righthand side game of Figure 3, for which we derived the Monte Carlo game tree as Figure 7. The corresponding decision tree is shown in Figure 10, where the probabilities are omitted, since these are unknown. In practice the "outcomes" would also be estimated. In this particular example, action 3 dominates action 1 regardless of the probabilities, whereas between actions 2 and 3, it is unclear which is better without the probabilities.

AlphaGo's use of Monte Carlo tree search is described on the first page of their 2016 *Nature* article in the following three excerpts (Silver et al. 2016, p. 484):

> ”Monte Carlo tree search (MCTS) uses Monte Carlo rollouts to estimate the value of each state in a search tree. As more simulations are executed, the search tree grows larger and the relevant values become more accurate. The policy used to select actions during search is also improved over time, by selecting children with higher values. Asymptotically, this policy converges to optimal play, and the evaluations converge to the optimal value function. The strongest current Go programs are based on MCTS..."

> "We pass in the board position as a 19×19 image and use convolutional layers to construct a representation of the position. We use these neural networks to reduce the effective depth

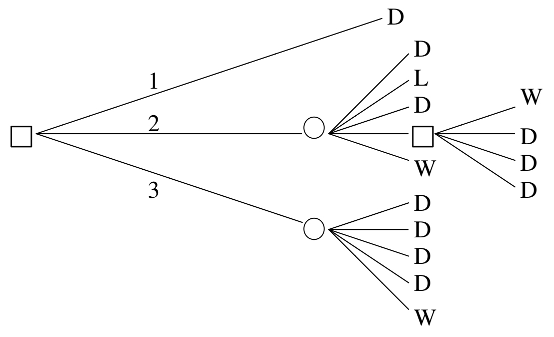

Figure 10: Decision tree for 2nd tic-tac-toe board configuration of Figure 3.

> and breadth of the search tree: evaluating positions using a value network, and sampling actions using a policy network."
>
> "Our program AlphaGo efficiently combines the policy and value networks with MCTS."

Figure 9 shows the corresponding $p(a|s)$ for the policy neural network that is used to simulate the opponent's moves and the value function $v(s')$ that estimates the value of a board configuration (state) using the value neural network. The latter could be used in the following way: if a state is reached where the value is known with sufficient precision, then stop there and start the backwards dynamic programming; else, simulate further by following the UCB prescription for the next move to explore.

The objective is to effectively and efficiently (through appropriate parametrization) approximate the value function approximation and represent the opposing player's moves as a probability distribution, which degenerates to a deterministic policy if the game is easily solved to completion assuming the opponent plays "optimally." However, at the core of these lies the Monte Carlo sampling of game trees or what our community would call the simulation of sample paths. Although the game is completely deterministic, unlike other games of chance such as those involving the dealing of playing cards (e.g., bridge and poker), the sheer astronomical number of possibilities precludes the use of brute-force enumeration of all possibilities, thus leading to the adoption of randomized approaches. This in itself is not new, even for games, but the Go-playing programs have transformed this from simple coin flips for search (which branch to follow) to the evaluation of a long sequence of alternating moves, with the opponent modeled by a probability distribution over possible moves in the given reached configuration. In the framework of MDPs, Go can be viewed as a finite-horizon problem where the objective is to maximize an expected reward (the territorial advantage) or the probability of victory, where the randomness or uncertainty comes from two sources: the state transitions, which depend on the opposing player's move, and the single-stage reward, which could be territorial (actual perceived gain) or strategic (an increase or decrease in the probability of winning), as also reflected in the estimated value after the opponent's move.

For *chess*, AlphaZero uses MCTS to search 80K positions per second, which sounds like a lot. However, Stockfish, considered in 2017 by many to be the best computer chess-playing program (way better than any human that has ever lived) considers about 70M positions per second, i.e., nearly three orders of magnitude more, and yet AlphaZero did not lose a single game to Stockfish in their December 2017 100-game match (28 wins, 72 draws). "Instead of an alpha-beta search with domain-specific enhancements, AlphaZero uses a general-purpose Monte-Carlo tree search (MCTS) algorithm." (Silver et al. 2017b, p. 3)

5 CONCLUSIONS

Monte Carlo tree search (MCTS) provides the foundation for training the "deep" neural networks of AlphaGo, as well as the single neural network engine for its successor AlphaZero. The roots of MCTS are contained in the more general adaptive multi-stage sampling algorithms for MDPs published by Chang et al. (2005) in *Operations Research*, where dynamic programming (backward induction) is used to estimate the value function in an MDP. A game tree can be viewed as a decision tree (simplified MDP) with the opponent in place of "nature" in the model. In AlphaZero, both the policy and value networks are contained in a single neural net, which makes it suitable to play against itself when the net is trained for playing both sides of the game.

The ML/AI community mainly views MCTS as a tool that can be applied to a wide variety of domains, so the focus is generally on the model that is used rather than the training method, i.e., the emphasis is on choosing the neural network architecture or regression model. Far less attention has been paid on the algorithmic aspects of MCTS. One direction that has started to be investigated is the use of ranking & selection (R/S) techniques for the selection of which path to sample next rather than traditional multi-armed bandit models, which tend to emphasize exploitation over exploration, e.g., leading to sampling the optimal move or action exponentially more often than the others. In fact, in his original paper introducing MCTS, Coulom himself mentions this alternative, viz., "Optimal schemes for the allocation of simulations in discrete stochastic optimization could also be applied to Monte-Carlo tree search" (Coulom 2006, p. 74), where one of the cited papers is the OCBA paper by Chen et al. (2000). Similar to R/S, in MCTS, there is also the difference between choosing which moves to sample further versus which move to declare the "best" to play at a given point in time. The distinction between simulation allocation and the final choice of best alternative in R/S is treated rigorously in a stochastic control framework in Peng et al. (2018), and again is noted by Coulom, viz., "in n-armed bandit problems, the objective is to allocate simulations in order to minimize the number of selections of non-optimal moves during simulations. This is not the objective of Monte-Carlo search, since it does not matter if bad moves are searched, as long a good move is finally selected. The field of discrete stochastic optimization is more interesting in this respect, since its objective is to optimize the final decision, either by maximizing the probability of selecting the best move, or by maximizing the expected value of the final choice." (Coulom 2006, p. 75) In the MDP setting, Li et al. (2018) investigate the use of OCBA in place of UCB for the sampling. They also note that the objective at the first stage may differ from that of subsequent stages, since at the first stage the ultimate goal is to determine the "best" action, whereas in subsequent stages the goal is to estimate a value function. Also, with parallel computing, it may actually make more sense to use batch mode rather than fully sequential allocation algorithms.

One caveat emptor when reading the literature on MCTS: Most of the papers in the bandit literature assume that the support of the reward is $[0,1]$ (or $\{0,1\}$ in Bernoulli case), which essentially translates to knowing the support. Depending on the application, this may not be a reasonable assumption. Estimating the variance is probably easier, if not more intuitive, than estimating the support.

As a final comment, note again that both Go and chess are deterministic games, in contrast to games such as bridge and poker, which involve uncertainty both in terms of future outcomes (which could also be viewed as how an opponent plays in Go and chess) and imperfect information (not actually knowing what cards the opponents hold). If the opponents' cards were known, then both bridge and poker would be relatively easy games to solve, since the number of possible outcomes is far lower than chess (not to mention Go). In these settings, it is the inherent uncertainty that makes the game difficult, but in January 2017, an AI system called Libratus, developed by Tuomas Sandholm and his PhD student Noam Brown at CMU, "defeated" (with 99.98% statistical significance based on 120,000 hands over 20 days) four of the world's top human players. As far as I could tell, the strategies that were developed followed more traditional AI approaches and did not use MCTS, but one possibility might be to convert the random game into a partially observable MDP (POMDP) and see if MCTS could be applied to the augmented state.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation (NSF) under Grant CMMI-1434419.

REFERENCES

AlphaGo Movie. 2018. https://www.alphagomovie.com, accessed August 12th, 2018.

Auer, P., N. Cesa-Bianchi, and P. Fisher. 2002. “Finite-time Analysis of the Multiarmed Bandit Problem”. *Machine Learning* 47(2-3):235–256.

Bertsekas, D. P., and J. N. Tsitsiklis. 1996. *Neuro-Dynamic Programming*. Belmont, MA: Athena Scientific.

Broadie, M., and P. Glasserman. 1996. “Estimating Security Price Derivatives Using Simulation”. *Management Science* 42(2):269–285.

Browne, C., E. Powley, D. Whitehouse, S. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. 2012. “A Survey of Monte Carlo Tree Search Methods”. *IEEE Transactions on Computational Intelligence and AI in Games* 4(1):1–49.

Chang, H. S., M. C. Fu, J. Hu, and S. I. Marcus. 2005. “An Adaptive Sampling Algorithm for Solving Markov Decision Processes”. *Operations Research* 53(1):126–139.

Chang, H. S., M. C. Fu, J. Hu, and S. I. Marcus. 2007. *Simulation-based Algorithms for Markov Decision Processes*. New York: Springer.

Chang, H. S., M. C. Fu, J. Hu, and S. I. Marcus. 2016. “Google Deep Mind’s AlphaGo”. *OR/MS Today* 43(5):24–29.

Chen, C.-H., J. Lin, E. Yücesan, and S. E. Chick. 2000. “Simulation Budget Allocation for Further Enhancing the Efficiency of Ordinal Optimization.” *Journal of Discrete Event Dynamic Systems: Theory and Applications* 10(3):251-270,

Coulom, R. 2006. “Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search”. In *Computers and Games: CG 2006*, edited by H. Jaap van den Herik et al., 72–83. Berlin, Germany: Springer.

DeepMind 2018. https://deepmind.com/research/alphago, accessed May 7th, 2017.

Fu, M. C. (ed). 2015. *Handbook on Simulation Optimization*. New York: Springer.

Fu, M. C. 2016. “AlphaGo and Monte Carlo Tree Search: The Simulation Optimization Perspective”. In *Proceedings of the 2016 Winter Simulation Conference*, edited by T. M. K. Roeder et al., 659–670. Piscataway, NJ: IEEE.

Fu, M. C. 2017. “Markov Decision Processes, AlphaGo, and Monte Carlo Tree Search: Back to the Future”. Chapter 4 in *Tutorials in Operations Research*, edited by R. Batta and J. Peng, 68–88. Catonsville, MD: INFORMS.

Gass, S.I., and M.C. Fu (eds). 2013. *Encyclopedia of Operations Research and Management Science*. 3rd ed. (2 volumes). New York: Springer.

Kahneman, D., and A. Tversky. 1979. “Prospect Theory: An Analysis of Decision Under Risk”. *Econometrica* 47(2):263–291.

Kocsis, L., and C. Szepesvári. 2006. “Bandit based Monte-Carlo Planning”. In *Machine Learning: ECML 2006*, edited by J. Fürnkranz et al., 282–293. Berlin, Germany: Springer.

Li, Y., M. C. Fu, and J. Xu. 2018. “Monte Carlo Tree Search with Optimal Computing Budget Allocation”. *2018 Conference on Neural Information Processing Systems (NIPS)*, December 3rd–8th, Montreal, Canada, submitted (under review).

Peng, Y., C.-H. Chen, E. K. P. Chong, and M. C. Fu. 2018. “Ranking and Selection as Stochastic Control”. *IEEE Transactions on Automatic Control* 63(8):2359-2373.

Powell, W. B. 2010. *Approximate Dynamic Programming*. 2nd ed. New York: Wiley.

Prashanth, L. A., C. Jie, M. C. Fu, S. I. Marcus, and C. Szepesvári. 2016. “Cumulative Prospect Theory Meets Reinforcement Learning: Prediction and Control”. In *ICML’16: Proceedings of the 33rd International Conference on Machine Learning*, edited by M. F. Balcan and K. Q. Weinberger, 1406–1415. New York: JMLR.org.

Silver, D., A. Huang, Aja, C. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. 2016. ”Mastering the Game of Go with Deep Neural Networks and Tree Search”. *Nature* 529 (28 Jan):484–503.

Silver, D., J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. 2017a. Mastering the Game of Go without Human Knowledge. *Nature* 550 (19 Oct):354–359.

Silver, D., T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. 2017b. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm", arXiv:1712.01815v1 [cs.AI] 5 Dec 2017.

Sutton, R. S. and A. G. Barto. 1998. *Reinforcement Learning: An Introduction*. Cambridge: MIT Press.

AUTHOR BIOGRAPHY

MICHAEL C. FU holds the Smith Chair of Management Science in the Robert H. Smith School of Business, with a joint appointment in the Institute for Systems Research and affiliate faculty appointment in the Department of Electrical and Computer Engineering, A. James Clark School of Engineering, all at the University of Maryland, College Park, where he has been since 1989. His research interests include stochastic gradient estimation, simulation optimization, and applied probability, with applications to manufacturing, supply chain management, and financial engineering. He has a Ph.D. in Applied Mathematics from Harvard and degrees in Mathematics and Electrical Engineering & Computer Science from MIT. He is the co-author of the books *Conditional Monte Carlo: Gradient Estimation and Optimization Applications*, which received the INFORMS Simulation Society's 1998 Outstanding Publication Award, and *Simulation-Based Algorithms for Markov Decision Processes*, and has also edited or co-edited four volumes: *Perspectives in Operations Research*, *Advances in Mathematical Finance*, *Encyclopedia of Operations Research and Management Science* (3rd edition), and *Handbook of Simulation Optimization*. He attended his first WSC in 1988 and served as Program Chair for the 2011 WSC. He has served as Program Director for the NSF Operations Research Program (2010–2012, 2015), *Management Science* Stochastic Models and Simulation Department Editor, and *Operations Research* Simulation Area Editor. He is a Fellow of INFORMS and IEEE. His e-mail address is mfu@umd.edu, and his Web page is https://www.rhsmith.umd.edu/directory/michael-fu.